Low Cost Debug Architecture using Lossy Compression for Silicon Debug

Ehab Anis and Nicola Nicolici
Department of Electrical and Computer Engineering
McMaster University, Hamilton, ON L8S 4K1, Canada
Email: anise@mcmaster.ca, nicola@ece.mcmaster.ca

Abstract

The size of on-chip trace buffers used for at-speed silicon debug limits the observation window in any debug session. Whenever the debug experiment can be repeated, we propose a novel architecture for at-speed silicon debug that enables a methodology where the designer can iteratively zoom only in the intervals containing erroneous samples. When compared to increasing the size of the trace buffer, the proposed architecture has a small impact on silicon area, while significantly reducing the number of debug sessions.

1. Introduction

For system-on-a-chip (SOC) designs, verification is a major contributing factor to the implementation cycle. As a consequence, significant work has been done in the area of *pre-silicon* verification, where design errors (or bugs) are caught through formal methods or extensive functional simulation [14]. The growing SOC complexity combined with the difficulty to accurately model integrated circuits (ICs), makes the pre-silicon verification methods inadequate to guarantee a product without errors before analyzing the first silicon. Furthermore, many electrical problems (e.g., leakage or drive fights) cannot be screened without the usage of the fabricated IC [11]. Consequently, due to the escalating mask costs it is imperative that the undetected functional and electrical bugs are fixed as soon as the first silicon is available. To achieve this challenge, *silicon debug*, which is the process of finding, locating and identifying design bugs in the *post-silicon* phase [2], is becoming a necessity.

1.1. Prior Work

To place the contribution of this paper in a larger context, this subsection discusses the main directions in silicon debug of digital ICs. Numerous *physical probing* techniques, such as electron or ion beam techniques, have been used extensively over the past few decades in IC failure analysis [19]. To deal with multiple metal layers (common to modern ICs), time resolved photoemission, that operates through the silicon substrate, has emerged [6]. Physical probing can also be used for screening electrical bugs, however, despite the recent advancements in this research field, the complexity of state-of-the-art devices requires a localization step to precede the destructive IC failure analysis. This step, which we call *logic probing*, correlates the simulation data to what is observed in the silicon (either on the input/outputs (I/Os) or on the internal signals, which are observed as discussed in the following paragraphs) in order to identify a subset of circuit nodes that need to be physically probed [20]. In addition, because many design errors in application specific integrated circuits (ASICs) are undetected functional bugs, physical probing is of no use for identifying them during the post-silicon phase. Moreover, logic probing is also essential for debugging designs mapped to field-programmable gate arrays (FPGAs). Therefore, due to their importance for finding both functional and electrical bugs, next we overview the logic probing techniques.

The debug of memories embedded in SOCs relies on the built-in self-test (BIST) or direct memory access infrastructure, which are commonly available on-chip for manufacturing test and diagnosis. However, whenever the above are not provided, as it is the case for small embedded memories (e.g., store buffers or instruction queues) methods that rely on clock control and the internal scan chains need to be employed [13]. The debug methods based on internal scan chains have been used extensively for debugging complex digital ICs [22]. In *scan-based debug*, the normal operation of the circuit is captured after the occurrence of a specific trigger event (or breakpoint). Subsequently the circuit state is observed by transferring it to the debug software either through the scan channels, if debugging on the automatic test equipment (ATE), or through the JTAG interface [18], when debugging in-field on the target application board.

One approach to reusing the scan logic for silicon debug is to integrate a debug module, which, in addition to controlling the breakpoints [21], it can start, stop, resume or single-step the execution. This approach obviously provides high observability of circuit behavior, nonetheless it is limited to the case when the input stimuli are *not* provided in real-time. An alternative approach that is applicable to real-time input stimuli, relies on stopping the functional test program when a failure is observed on an output and subsequently, by re-running the debug experiment with

978-3-9810801-2-4/DATE07 © 2007 EDAA

different trigger points, it will scan the state before and after the observed failure point. Thereafter, post-processing algorithms, such as latch divergence analysis [4] or failure propagation tracing [3], can be used to analyze the scan dumps and identify the first failing state elements. This approach, nevertheless, needs to stop the circuit execution after a failing point is identified, thus ignoring any further erroneous behaviors. Because hard-to-detect functional bugs appear in circuit states which may be exercised billions of cycles apart [11], it is therefore desirable to continue the execution after a failing point without halting. However, to complete a scan dump while continuing the real-time execution, it is necessary to double buffer the state elements in the scan chain, which will likely lead to unacceptable area penalty [12]. A *complementary* approach to scan is to monitor only a subset of internal signals that are dumped in real-time in a *trace buffer*. While scan-based debug concepts have emerged from the manufacturing test research, trace buffer-based debug, which is discussed next, has been influenced by software debugging used in embedded systems [16].

Real-time systems centered around embedded processors or micro-controllers and have been traditionally debugged using in-circuit emulator (ICE) devices. ICEs are constructed using bond-out chips, which connect internal nodes to additional device I/Os in order to make them visible *off-chip* to external instruments. The limitation of using ICEs for state-of-the-art SOC devices lies not only in the increasing gap between the on-chip and off-chip frequencies, but also in a large footprint of the bound-out chips caused by the additional I/Os used only for debug. As a consequence, for SOC designs (both FPGA and ASIC implementations) there has been a trend toward placing the instrumentation *on-chip*, thus enabling at-speed sampling through embedded logic analysis [10, 17]. The sampled data is subsequently sent via a low bandwidth interface from the internal debug module to the external debug software for post-processing. The *trace buffer-based debug* methods can be broadly classified as: *special-purpose* (i.e.., specific to embedded processors) [8, 9] or *generic* (i.e., applicable to any type of custom SOCs) [1, 15]; with *centralized* tracing where one trace buffer is used per SOC (with different interconnect topologies between the embedded cores and the trace buffer) [1, 8, 9] or *distributed* sampling with trace buffers allocated to individual cores [15]. Regardless wether they are special-purpose, generic, centralized or distributed, the distinguished benefit of the trace buffer-based methods is the ability to do in-field at-speed debug, which is also suitable for analyzing the no-trouble-found parts [5].

1.2. Motivation and Objective

Given the in-field use and at-speed sampling advantages of trace buffer-based debug methods, an obvious question is what kind of drawbacks are limiting their use? The functional bugs that are hard-to-detect will manifest themselves only several times over a large execution time [11]. Therefore, to accelerate the identification of the root of a problem, one would like to correlate the few different failing observations that occur while running a long debug experiment. To achieve this, the *observation window* of an experiment needs to be extended, however, given the limited on-chip area available for trace buffers, the number of *debug sessions* (i.e., the number of times the same experiment is rerun with different trigger points that initiate on-chip sampling) may increase significantly. Therefore, the answer to the above question lies in the trade-off between the trace-buffer area and the number of debug sessions for a given observation window. The trace-buffer area is determined by its *width*, which constrains the number of signals to be probed, and its *depth*, which limits the number of samples to be stored. To extract as much data as possible from a given debug session, without increasing the on-chip area, *compression* for the width and depth of the trace buffer can be employed. This idea has been explored in different ways for both special-purpose and generic scenarios.

On the one hand, for the special-purpose case (e.g., [8]) the compression solutions are well established [10, 16]. The key idea for width compression is to probe a subset of signals from which the state of the embedded processor can be reconstructed as accurately as possible (e.g., pipeline status signals or indirect branch signals). The depth compression is accomplished by eliminating the redundant temporal information on data or address busses (e.g., through differential compression) or by setting up filters throughout the observation window that enable selective sampling in time, which helps avoid trace buffer overflows. On the other hand, for the generic case the compression solutions are currently emerging. For example, width compression can be achieved by automatically analyzing the design and identifying a subset of essential signals, which, after being captured on-chip, will be ”inflated” off-line using the knowledge of the design [1]. Motivated by the fact that, to the best of our knowledge, there are no methods (reported in the public domain) for depth compression applicable to any design, it is the aim of this paper to investigate this problem.

2. Proposed Iterative Debug Framework

The method proposed in this paper is applicable to the case where the debug data is known a-priori and a deterministic execution of input data will always produce the same output data. This is the case when debugging on an ATE and it is also common on a target application board where stimuli are applied synchronously (e.g., audio/video decoder) and the expected responses can be computed using a reference behavioral model of the circuit under debug (CUD). This debugging method is also referred to as *cyclic debugging* or *deterministic replay* in the software engineering literature and having an experimental testbed that supports it is a pre-requisite for the iterative debug flow described next.

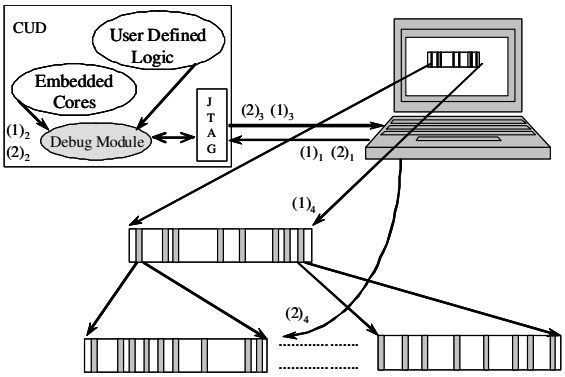

Figure 1. The Iterative Debug Flow

The basic intuition of our approach can be explained as follows. In cyclic debugging, by re-running the same experiment deterministically, the use of different debug module configurations (such as deciding to start sampling at distinct times in different debug sessions) can facilitate the reconstruction of the circuit behavior over a long observation window. The straightforward way of achieving this is resume sampling in a new debug session from the same point in time where the previous debug session has stopped. However, if the trace buffer on the chip is small and the targeted observation window is long, this will lead to unnecessarily large number of debug sessions. Therefore, we propose to run the first debug session in a *compressed mode* and learn from it which *intervals* in the observation window are error-free. This can be achieved through *lossy compression* by mapping intervals onto *signatures*. The targeted observation window and the size of the on-chip trace buffer will determine the *compression level*. As shown in Figure 1 (where $(i)_j$ means step j in debug session i), the debug software will prepare and load the CUD configuration data ($(1)_1$) for the first debug session. Then, the debug experiment is run on-chip and the compressed intervals (i.e., signatures) will be sampled into the trace buffer ($(1)_2$, note the hardware required to achieve this will be discussed in the following section). Then the trace buffer content will be offloaded to the debug software ($(1)_3$), where the failing signatures (and hence intervals) will be identified ($(1)_4$). If a signature is error-free, then, in the remaining debug sessions, the interval whose sequence of samples maps onto the respective signature, will be skipped and no samples will be extracted from it. In the following debug session the same steps are completed ($(2)_1$, $(2)_2$, $(2)_3$ and $(2)_4$). Note, however, using the failing information from the preceding debug session the user can set up the CUD configuration such that he *zooms-in* only into the failing intervals. This process is repeated *iteratively* in the succeeding debug sessions until all the failing intervals are extracted. The benefits stem from the fact that we can observe the failing behaviors within a long observation window without wasting any debug sessions for sampling the error-free intervals. The debug architecture that facilitates this iterative debug flow is described next.

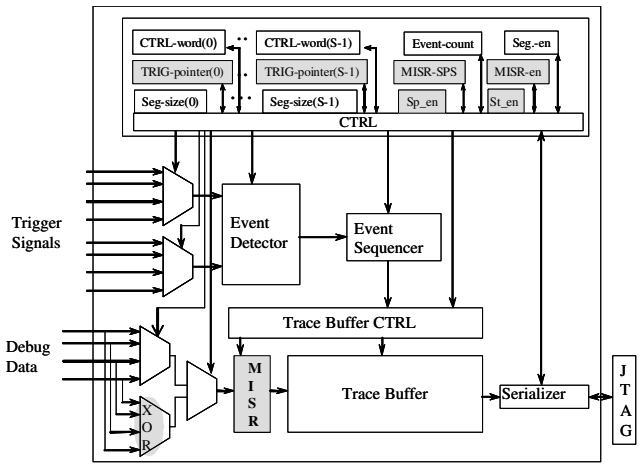

Figure 2. The Embedded Debug Module

3. Debug Architecture for Lossy Compression

This section describes the distinguishing features of the proposed debug architecture. We start by explaining the common features in an embedded debug module. This is followed by the hardware modifications required to support the iterative debug flow based on lossy compression.

3.1. Embedded Debug Module

As shown in Figure 2, at the core of a debug module used for embedded logic analysis is an event detector. Its purpose is to monitor a group of trigger signals so it can determine when the data signals will be sampled in the trace buffer. In our implementation triggering can be done based on bitwise, reduction, comparison and logical operations between the trigger signals. Three levels of operators can be combined and the operations can be performed between two trigger signals selected by the control word ($CTRL - word$). Since the event detector contains many levels of logic, if necessary, its speed can be improved to support higher sampling frequencies by pipelining it. To enable sequential event detection, based on the configuration of the control words and the $Event - count$ field (which specifies the number of trigger events), the event detector is interfaced to an event sequencer. When the segmented mode flag ($Seg - en$) is enabled, the trace buffer is divided into multiple S segments. Note, however, in this mode a $CTRL - word$, as well as a segment size ($Seg - size$), needs to be supplied (as part of the CUD configuration data provided for a debug session) for each of the S segments. After the trace buffer is filled with samples, its content is serialized and sent back to the off-chip debugger software via a low-bandwidth interface such as JTAG. If the stream mode is enabled ($St - en$) the debug data will be streamed as it is sampled (provided that the trace buffer has one read and one write port). This will increase the number of samples acquired in one debug session, nonetheless the trace buffer will eventually become full, since the bandwidth for streaming is limited.

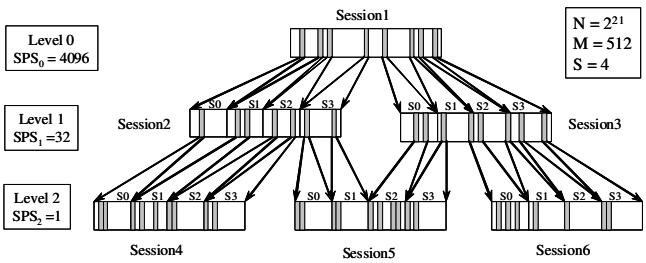

Figure 3. An Illustrative Debug Example

3.2. MISR-Based Lossy Compression

Our proposed solution relies on a multiple input signature register (MISR) (commonly used in BIST), which is placed at the input of the trace buffer. This MISR performs lossy compression on the selected debug data signals with a compression level determined by the *samples per signature* ($MISR{-}SPS$) parameter, whenever the MISR enable $MISR-en$ is enabled. At the beginning of a debug session, the MISR counter (which is placed in the controller of the debug module) is initialized to zero. After the trigger condition occurs, the MISR counter starts counting and it resets back to zero each time it reaches $MISR-SPS$, at which point the signature is written in the trace buffer. The number of signatures stored in each segment is determined by $Seg-size$. After the first debug session, in order to start sampling at the beginning of an interval of interest, a trigger pointer ($TRIG-pointer$) is used for each segment. The value of this trigger pointer is pre-computed off-line (based on the failing information from the previous debug level) and it is also passed as part of the CUD configuration data at the beginning of each debug session. When all the failing intervals of length smaller than the segment size have been identified, the compression does not need to be performed any more and $MISR-en$ is deactivated.

Due to space limitations we cannot elaborate on all the steps, algorithms and equations used to update the CUD configuration data from one debug session to another. As a consequence, we use Figure 3 and the following example to show how the features described in this section can be used for the proposed iterative debug flow.

Example 1 *We assume the trace buffer has $S = 4$ segments and its depth is $M = 512$ locations, and the targeted observation window is $N = 2^{21}$ samples. We start with an initial debug session that compresses the entire observation window with the compression level given by $SPS_0 = N/M = 4096$ (SPS_i is the number of samples per signature at debug level i). Debug level 0 has only one debug session, in which the trace buffer is unsegmented. After the initial debug session, each of the eight failing signatures covers 4096 samples, which, for this particular case, is greater than the size of the trace buffer (512 locations). Therefore, we will have an intermediate debug level 1, which will further "filter-out" the error free intervals. For all the subsequent debug sessions, the memory segments are used in such way that instead of expanding one failing signature per debug session, S failing signatures are expanded. SPS_1 can be calculated as $SPS_0/(M/S) = 32$. The trigger pointers in each debug session are calculated based on the SPS of the current debug level and the starting point of the failing signatures from the previous debug level. In the last debug level 2, no compression will be applied ($SPS_2 = 1$). To better utilize the available memory, before running any session in debug level 2, the debugger software will check if any neighboring signatures from level 1 can be merged such that more than one signature can be expanded into one segment in debug level 2. In Figure 3, two of the failing signatures from session 2 at debug level 1 can be expanded in one segment in debug level 2. Therefore, since the number of erroneous intervals is only 12 (it equals the number of segments with failing samples in the last debug level) the number of debug sessions for the proposed approach is 6. In the sequential debug case (i.e., no compression), the total number of sessions to collect data over the same observation window would be $N/M = 4096$.*

If the number of failing samples is high then the proposed method will unlikely bring any benefits because iteratively we will have to zoom in a large number of intervals. However, for low error rates, which would be a realistic assumption when searching for *hard-to-detect* bugs that occur spuriously in large observation windows [11], the reduction in the number of debug sessions is considerable as shown in the experimental section.

3.3. The Addition of Spatial Compression

The $Sp-en$ flag shown in Figure 2 is used to enable spatial compression. In this case width compression can be employed before the MISR by using an XOR network in which multiple channels of debug data are compressed into a single one. This feature can be used in all the debug levels except the last one, where no compression is performed and the debug module selects only one channel at a time. Note, while both the spatial compressor and the MISR may run into the aliasing problem it is unlikely that all the erroneous samples caused by a particular bug will lead fault-less signatures in all the possible intervals of occurrence.

3.4. Combining Streaming and Compression

An interesting observation is that if the length of the observation window is large and if the failing signatures are sparse, then there is plenty of idle time in between any two trigger points in the trace buffer. Therefore, an additional architectural feature ($St-en$ flag in Figure 2) enables the streaming of the samples stored in the trace buffer through the JTAG port while the debug experiment is still running on the CUD. The basic principle can be explained as follows.

We define the *streaming distance* as the minimum distance between two trigger points that can be mapped onto

the first segment of the trace buffer, without overflowing it when the stream mode is enabled. The streaming distance is calculated based on the on-chip sampling frequency, the off-chip serial interface frequency and the trace buffer segment size. While streaming out the contents of the first segment, another trigger pointer that satisfies the streaming distance can be uploaded to the debug module. The process of streaming out the first segment contents and uploading a new trigger pointer for the same segment is repeated until the last $S-1$ signatures in the current debug level are reached. Subsequently, after offloading the content of the trace buffer to the debugger software, the above steps are repeated iteratively until all the failing intervals are detected.

4. Experimental Results

This section discusses the experiments concerning the area investment and compression benefits of the iterative debug flow presented in this paper. It should be noted, the proposed solution has not been implemented into an ASIC. Nonetheless, the area results have been estimated using a 180nm ASIC standard cell library. The rest of the experiments have been done using the data available from the FPGA prototyping of an MP3 audio decoder [7] or using random data, as explained later in this section.

4.1. Area of the Proposed Debug Module

Table 1 shows the area of the debug module (without the trace buffer) in terms of 2 input NAND (NAND2) gates. The results shown are for different number of variable-sized segments ($S=2$, $S=4$ and $S=8$) for the following cases: no compression and no high speed sampling features; no spatial compression ($Ch=1$) and spatial compression with different number of channels ($Ch=2$, $Ch=4$ and $Ch=8$) where streaming compression is enabled and high speed sampling is facilitated by pipelining the event detector. The results from the table refer *only* to the logic area and do not account for the trace buffer. To ensure that the debug module can be shared between multiple logic cores on the SOC, we connect 8 different groups of signals to the same debug module. In addition, it is important to have as many features as possible in the control word (such as compare against a constant or mix several types of logic or relational operators), the control word (which is stored for each segment) will grow in size. Should all these features be removed, the area of the debug module can be significantly reduced, nevertheless the debug capabilities will be severely limited, which we do not consider to be a good motivation.

The different variants of the proposed architecture are obviously larger than the debug architecture that does not have any compression or high-speed features. What is interesting to note, however, as the number of channels used for spatial compression increases, the added area becomes insignificant. Nonetheless, as shown later in this section spatial compression feature can further reduce the number of debug sessions. It is essential to note also that the increase in the logic area of the debug module has significantly less penalty than scaling the trace buffer. For example, for 8 segments and 8 channels that are spatially compressed, the added logic area for compression is still less than one fifth of the size of an embedded memory of 4Kbytes implemented in the same technology.

Segments Number	No Compression	Compression			
		$Ch=1$	$Ch=2$	$Ch=4$	$Ch=8$
$S=2$	2318	4585	4649	4725	4880
$S=4$	3333	6169	6234	6314	6468
$S=8$	5359	9044	9108	9188	9343

Table 1. Area in NAND2 Equivalents

Error %		No Streaming		Streaming	
		No Spatial	Spatial	No Spatial	Spatial
0.164	T_{seq}/T_{prop}	45.0	59.7	76.2	111.6
1.590	T_{seq}/T_{prop}	6.8	8.7	12.3	18.0
6.143	T_{seq}/T_{prop}	2.4	3.1	4.6	6.9

Table 2. Reduction in Debug Execution Time for MP3 ($N=2^{21}$, $M=512$, $S=4$, $Ch=2$)

4.2. MP3 Decoder Experiments

An MP3 decoder has been implemented and prototyped on an FPGA board. It was investigated how functional errors in the RTL code, can be detected using the proposed methodology. Table 2 shows the reduction ratios in terms of the *debug execution time* for the entire observation window, where T_{seq} and T_{prop} are the debug execution times for the sequential (i.e., no compression) and the proposed debug methods respectively. To compute the debug execution time we need to consider not only the number of the debug sessions, but also the *on-chip sampling time* and *the communication time* (needed for off-loading the trace buffer content to the debugger software through the JTAG interface) for *each of the debug sessions*. On the one hand, the communication time is determined by the JTAG frequency and the capacity of the trace buffer and therefore it is constant for all the debug sessions. On the other hand, for the proposed method the *on-chip sampling time* is dependent on how many on-chip clock cycles elapse from the trigger event until the trace buffer is filled only with failing intervals, which are of interest in the current debug session (e.g., in Figure 3 the on-chip sampling time for debug session 3 will be larger than for debug session 2). Therefore, this time varies from one debug session to another and it is dependent on the distribution of the failing samples in the observation window. It should be noted that the debugger software does not incur any additional latency because the processing of debug data can be done at the same time while the debug experiments are running on-chip and/or the data is transferred to/from the debugger software.

The data reported in Table 2 is for probing the data busses after the stereo decoder module in the MP3 decoder's pipe [7] for 3 different functional bugs affecting only several stereo modes. The error choices are motivated by the fact that only a few music frames throughout an entire song

Error %	Burst length	64	128	256	512
1.553	T_{seq}/T_{prop}	7.1	11.9	18.3	26
2.324	T_{seq}/T_{prop}	5	8.3	13	18.4
3.077	T_{seq}/T_{prop}	3.9	6.5	10.1	14.4
3.831	T_{seq}/T_{prop}	3.2	5.3	8.3	11.8

Table 3. Reduction in Debug Execution Time versus Error % of Random Data for Different Burst Lengths with $N = 2^{27}$, $M = 2048$, $S = 4$

will use a specific stereo mode, thus justifying the condition that the bugs are very difficult to find and they manifest themselves only spuriously over very long observation windows (the error rates are .16%, 1.59% and 6.14% respectively). In this particular case, the observation window represents 1820 MP3 frames (there are 1152 samples per MP3 frame), which gives $N = 2^{21}$. The spatial compression is applied on both music channels of the MP3 decoder and, as clearly shown in Table 2, streaming can bring substantial improvements. The significant reduction in the debug execution time when using streaming is due to the fact that by using the proposed iterative debug flow, by sampling only the failing intervals from the previous debug level we can use the time in between two intervals to stream out samples that have been sampled in the same debug session.

4.3. Random Data Experiments

To show the sensitivity of the results on the distribution of failing samples, Table 3 shows the reduction in debug execution time for different sets of random data experiments that have distinct error distributions. The random data experiments assume $N = 2^{27}$, $M = 2048$, $S = 4$, and that no spatial compression and no streaming compression are enabled. The burst length represents the number of erroneous samples that occur consecutively and all the bursts are randomly distributed over the entire observation window. As it can be clearly observed from the table, when the burst length increases, the reduction ratios grow. This is because when the burst length is approaching the segment size, for a given error percentage, the number of uncompressed debug sessions (the last level of debug) are reduced (also the number of debug sessions in the intermediate levels are indirectly reduced as less failing signatures need to be processed). Finally, as expected, as the error percentage increases, the reduction ratios decrease for the same burst length since more debug sessions will be necessary to detect the erroneous intervals.

5. Conclusion

This paper has shown how lossy compression can be used for developing a new debug architecture and an iterative debug flow that enable an increase in the observation window while supporting the existing depth or width compression techniques for at-speed silicon debug of generic digital ICs . The proposed solution can be used as an aid to the existing methods for trace buffer-based debug, whenever cyclic debugging is permitted.

References

[1] M. Abramovici and Y.-C. Hsu. A New Approach to Silicon Debug. In *IEEE International Silicon Debug and Diagnosis Workshop (SDD)*, November 2005.

[2] M. Abramovici, E. J. Marinissen, M. Ricchetti, and B. West. Suggested Terminology Standard for Silicon Debug and Diagnosis. In *IEEE International Silicon Debug and Diagnosis Workshop (SDD)*, November 2005.

[3] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor Silicon Debug Based on Failure Propagation Tracing. In *Proceedings IEEE International Test Conference (ITC)*, pages 284–293, October 2005.

[4] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *Proceedings IEEE International Test Conference (ITC)*, pages 755–763, October 2003.

[5] S. Davidson. Understanding NTF Components from the Field. In *Proceedings IEEE International Test Conference (ITC)*, pages 333–342, October 2005.

[6] R. Desplats, F. Beaudoin, P. Perdu, N. Natara, T. Lundquist, and K. Shah;. Fault Localization Using Time Resolved Photon Emission and STIL Waveforms. In *Proceedings IEEE International Test Conference (ITC)*, pages 254–263, October 2003.

[7] S. Hacker. *MP3: The Definitive Guide*. O'Reilly & Associates, Inc., Mar. 2000.

[8] A. Hopkins and K. McDonald-Maier. Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores. *IEEE Transactions on Computers*, 55(2):174–184, February 2006.

[9] Y. Huang and W.-T. Cheng. Using Embedded Infrastructure IP for SOC Post-Silicon Verification. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 674–677, June 2003.

[10] IEEE Industry Standards and Technology Organization. *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. http://www.nexus5001.org, 2003.

[11] D. Josephson. The Manic Depression of Microprocessor Debug . In *Proceedings IEEE International Test Conference (ITC)*, pages 657 – 663, October 2002.

[12] D. Josephson and B. Gottlieb. The Crazy Mixed up World of Silicon Debug. In *Proceedings IEEE Custom Integrated Circuits Conference (CICC)*, pages 665 – 670, October 2004.

[13] Y.-J. Kwon, B. Mathew, and H. Hao. FakeFault: A Silicon Debug Software Tool for Microprocessor Embedded Memory Arrays. In *Proceedings IEEE International Test Conference (ITC)*, pages 727–732, October 1998.

[14] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[15] R. Leatherman and N. Stollon. An Embedded Debugging Architecture for SOCs. *IEEE Potentials*, 24(1):12–16, February 2005.

[16] C. MacNamee and D. Heffernan. Emerging On-chip Debugging Techniques for Real-Time Embedded Systems. *IEE Computing & Control Engineering Journal*, 11(6):295–303, December 2000.

[17] K. Morris. On-Chip Debugging - Built-in Logic Analyzers on your FPGA. *Journal of FPGA and Structured ASIC*, 2(3), January 2004.

[18] K. Parker. *The Boundary-Scan Handbook : Analog and Digital*. Kluwer Academic Publishers, 2nd edition, 1998.

[19] J. Solden and R. Anderson. IC Failure Analysis: Techniques and Tools for Quality and Reliability Improvement. *Proceedings of the IEEE*, 81(5):703–715, May 1993.

[20] D. Vallett. IC Failure Analysis: The Importance of Test and Diagnostics. *IEEE Design and Test of Computers*, 14(4):76–82, July 1997.

[21] B. Vermeulen, M. Urfianto, and S. Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 514–517, June 2004.

[22] B. Vermeulen, T. Waayers, and S. K. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proceedings IEEE International Test Conference (ITC)*, pages 638 – 647, October 2002.